Programmieren
mit
C

aus: *ComputerKurs*-Magazin, Jahrgänge 1984/1985

C wie Computer

Programmiersprachen lassen sich in zwei Hauptgruppen unterteilen: Hochsprachen wie BASIC oder PASCAL und maschinennahe Sprachen. Die Sprache C überbrückt die Kluft zwischen beiden Gruppen.

In den letzten Jahren wurde mit viel Energie an Programmiersprachen gearbeitet, die menschliche Denkvorgänge imitieren und den Programmierer von den Eingrenzungen der Hardware freihalten sollten. Trotz aller Anstrengungen ist aber immer noch viel Programmierarbeit nötig, wenn Hardware direkt gesteuert werden muß, damit kleine und relativ langsame Maschinen maximale Leistung erzielen können.

Anpassungen dieser Art werden üblicherweise in der Assemblersprache vorgenommen. Gelegentlich finden auch Sprachen Anwendung (zum Beispiel FORTRAN), die auf eine bestimmte Hardware abgestimmt sind. Assemblersprachen erreichen zwar (geschickt programmiert) maximale Wirkung, sind aber nur auf einen Maschinentyp ausgerichtet. Es bestand daher schon seit langem der Bedarf für eine einfache Programmiersprache auf relativ niedriger Ebene, die auf einem breiten Maschinenspektrum läuft und die Strukturen und Vorteile der Hochsprachen mit der direkten Hardwaresteuerung verbindet.

BCPL war die erste Sprache, die diesen Anforderungen entsprach. Sie wurde von Martin Richards an der Cambridge University entwikkelt. Als Ken Thompson und seine Mitarbeiter in den Bell Laboratorien am Betriebssystem Unix arbeiteten, suchten sie dafür eine Hochsprache und übernahmen viele Konzepte von BCPL in eine Sprache namens B. Kernighan und Ritchie entwickelten aus B schließlich C und veröffentlichten die Sprachdefinition 1978 in dem Buch „The C Programming Language", das als Standard für C-Anwendungen gilt – bis letztlich ein neuer internationaler Standard entwickelt wird.

C und Unix standen immer in enger Verbindung. Von den 13 000 Codezeilen, aus denen die Kernroutinen von Unix bestehen, wurden nur etwa 800 Zeilen in Assembler entwickelt, der Rest entstand in C. Durch ihren geringen Befehlsumfang ist C leicht erlernbar. Sie vereinfacht außerdem die Erstellung von Compilern und läßt sich damit leicht auf viele verschiedene Maschinentypen anpassen. Da C sehr maschinennah ist, ist auch ihr Code kompakt und schnell. Es wurden damit nicht nur Unix und Unix-Anwendungen geschrieben, sondern auch größere Teile von MS-DOS und CP/M erstellt.

C-Compiler gibt es für fast alle Micros und auch für viele Maschinen der Groß-EDV. Softwarehäuser können unter C erstellte Programme leicht auf völlig verschiedenartige Geräte wie den Apple Macintosh und IBM PC übertragen. In dieser Serie halten wir uns an die Version von Kernighan und Ritchie und setzen – beispielsweise bei vorgefertigten Funktionen – das Betriebssystem Unix voraus.

C-Programme werden aus anwenderdefinierten Funktionen aufgebaut, die eine Reihe von Argumenten, Werten oder Pointer auf Werte annehmen können und Werte oder Pointer zurückliefern (die weiterverarbeitet oder ignoriert werden können). Jede Funktion wird über ihren Namen aufgerufen.

Die Funktion „main"

Alle C-Programme beginnen mit der Ausführung der Funktion „main". Die äußerste Programmebene muß daher eine Definition dieser Funktion sein. Das folgende Programm ist zwar sehr einfach, aber dennoch vollständig:

```
main()
{
    printf("hallo Welt/n");
}
```

Es zeigt einige wichtige Eigenschaften der Sprache. Die Definition einer Funktion besteht aus einem Funktionsnamen, gefolgt von einer in Klammern stehenden Argumentenliste. In diesem Beispiel hat main zwar keine Argumente, muß aber Klammern enthalten.

Der Hauptteil der Funktionsdefinition besteht aus einer Folge von Anweisungen (in diesem Fall gibt es nur eine), die von Klammern ([]) umschlossen sind und mit einem Semikolon enden. Die Klammern ähneln dem „Begin . . .End" von PASCAL. Sie umschließen eine beliebige Folge von Befehlen oder Deklarationen, die nun wie einzelne Befehle eingesetzt werden können. Der Gebrauch des Semikolons ist jedoch einfacher als PASCAL – jede vollständige Anweisung wird durch ein Semikolon abgeschlossen. Unser Programm enthält nur eine einzige Anweisung – die Standardfunktion printf.

Die Ein- und Ausgabe wurde in C nicht fest definiert, da sie von Gerät zu Gerät stark variieren kann. Statt dessen liefert jede Anwendung einen eigenen Satz von E/A-Funktionen, die dem Sprachstandard entsprechen müssen. Der von printf zurückgelieferte Wert hat keine

Hisoft-C läuft auf Schneider und Spectrum Computern. Die Version hält sich eng an den Standard, verfügt jedoch nicht über das Fließkommaformat. Hisoft hat jedoch eine Fließkommaversion angekündigt, die in Kürze herauskommen soll. Der einzige andere Nachteil ist das Handbuch. Es ist zwar umfassend angelegt, aber so unübersichtlich, daß es sich für Anfänger nicht eignet. Wenn Sie jedoch schon Erfahrungen mit der Sprache haben oder über ein anderes Lehrbuch verfügen, ist dieses Paket sehr brauchbar.

Escape-Zeichen

\n Neue Zeile
\t Horizontales Tab
\b Rückwärtsschritt
\r Return
\f Seitenvorschub
\' Gibt ein Apostroph aus (wird als Zeichenbegrenzung eingesetzt)
\" Gibt ein Anführungszeichen aus (wird als Stringbegrenzung eingesetzt)
\d d ist eine Oktalstelle; gibt das Zeichen mit dem entsprechenden ASCII-Code aus
\xh h ist eine Hexadezimalstelle; gibt das Zeichen mit entsprechendem ASCII-Code aus
\e Escape
\v Vertikales Tab

Formatumwandlung

Zeichen	Argumenttyp	Bedeutung
d	int	Dezimalformat
o	int	Oktalformat
x	int	Hexadezimalformat
u	int	Dezimalformat ohne Vorzeichen
c	char/int	Ausgabe als einzelnes Zeichen
s	*char	Ausgabe als String
e	float/double	Mantisse-Exponent-Format
f	float/double	Normales Dezimalformat
g	float/double	das kürzeste Format von d, e oder f
%		Ausgabe des %-Zeichens

Bedeutung und wird ignoriert. printf kann eine beliebige Anzahl von Argumenten haben: numerische, Zeichenvariablen oder Strings (siehe oben). Auch gibt printf nicht automatisch ein Return aus, sondern nur auf Anweisung. C hat dafür spezielle Steursymbole, die mit dem „Backslash" (\) beginnen. Das Symbol für Return ist \n (siehe Tabelle der Escape-Steuerzeichen).

Das folgende Programm erzeugt das gleiche Ergebnis wie unser erstes Beispiel:

```
main()
{
      printf("hallo");
      printf("Welt");
      printf("/n");
}
```

C arbeitet mit Typen – das heißt, der Typ jeder Variable muß deklariert werden. Folgende Typen sind ständig vorhanden:

char – Zeichen, die ein Byte belegen
short – kurze Ganzzahlen
int – normale Ganzzahlen
long – ganze Ganzzahlen
float – reelle Zahlen im Fließkommaformat
double – Zahlen im Fließkommaformat mit doppelter Genauigkeit

Variablendeklarationen bestehen aus einem Typennamen, gefolgt von einer Liste sämtlicher Variablennamen dieses Typs. Alle Variablen müssen deklariert werden, wobei sich die Deklarationen an einer beliebigen Stelle des Programms befinden können. Variablennamen dürfen beliebig lang sein, jedoch werden nur die ersten acht Stellen verarbeitet.

Wie auch in anderen Sprachen dürfen Variablennamen in C keine Leerzeichen enthalten, wohl aber das Zeichen für Unterstreichung (Underline). Die Namen dürfen weiterhin nicht mit einer Zahl oder einem Underline beginnen. Groß- und Kleinbuchstaben werden unterschieden: A ist eine andere Variable als a. Die Bytezahl der numerischen Typen ist nicht festgelegt und kann je nach Anwendung variieren. Eine Ganzzahl vom Typ „short" hat normalerweise acht Bits, „int" 16 Bits und „long" 32 Bits.

Zuordnungen und mathematischen Abläufe folgen den üblichen Regeln, wobei das Gleichheitszeichen (=) die Rolle des Zuordnungsoperators spielt. Mit % eingesetzt, ergeben die arithmetischen Operatoren +, –, * und / den Modulowert:

```
int x,y,z
...
z=x%y
```

Ausdrücke und Zuordnungen können beliebig viele numerische Typen und „char" enthalten, wobei char hierbei als Ein-Byte-Ganzzahl (mit dem ASCII-Code eines Zeichens) angesehen wird. C führt alle Typenumwandlungen von „niedrigen" in „höhere" Typen automatisch aus. Hierbei ist interessant, daß normalerweise alle Operationen im Fließkommaformat in doppelter Genauigkeit ausgeführt werden, selbst wenn alle Operanden nur vom Typ „float" sind. Typenänderungen lassen sich aber auch forcieren, indem der neue Typ in Klammern vor dem Variablennamen aufgeführt wird:

```
int n;
float f;
...
f=sqrt((double)n);
```

übernimmt eine Kopie des Wertes n in doppelter Genauigkeit, da die Quadratwurzelfunktion als Argument den Type double erwartet. Der Inhalt von n bleibt dabei unverändert.

Mit ++ und –– besitzt C zwei weitere praktische Zusatzoperationen für die In- und Dekrementierung. ++a inkrementiert den Wert a und, ––a dekrementiert ihn, bevor der aktuelle Befehl ausgeführt wird. Bei a++ und a–– geschieht die In- und Dekrementierung erst nach Beendigung des aktuellen Befehls.

```
int a,b
...
b=1;
a=b++; /*ergibt 1 in a und 2 in b*/
```

während

```
b=1
a=++b;/* die 2 sowohl in a und b ergibt */
```

Da diese Zuordnung als Funktion behandelt wird, liefert sie auch ein Ergebnis zurück: den zugeordneten Wert. Dieser Wert läßt sich nun in einem anderen Ausdruck wie

```
x=y=z;
```

weiterverarbeiten. Das Beispiel ordnet x und y den Wert von z zu.

Wir hatten bereits erwähnt, daß Ein- und Ausgabe von Standardfunktionen ausgeführt werden, die je nach Anwendung variieren können. Da es jedoch schwierig ist, Programme ohne ein E/A-Format zu schreiben, gehen wir am Ende dieser Folge noch kurz auf die beiden Hauptfunktionen der Terminal-E/A ein. Eine kennen wir bereits: printf formatiert Werte der verschiedensten Typen und sendet sie an das Standard-Ausgabegerät (normalerweise den Bildschirm). Das vollständige Format sieht so aus:

```
printf(control-string, arg1, arg2, .....);
```

Argumente können Strings oder Variablen der grundlegenden Datentypen sein.

Der „Control-string" kann normale Zeichenfolgen enthalten, die ohne Veränderung ausgegeben werden. Er hat jedoch auch spezielle Formatierungsangaben für jedes der Argumente und legt auch die Anzahl der Argumente fest. Formatierungsangaben beginnen mit % und enden mit einem der Umwandlungszeichen (siehe Tabelle). Zwischen diesen beiden Zeichen bestimmt das Zeichen „–" die Linksbündigkeit des Ausgabefeldes, eine Zahl jedoch die Feldbreite einer Zahlenausgabe. Ein Punkt gefolgt von einer weiteren Zahl legt (bei einem String) die Anzahl der anzuzeigenden Zeichen fest oder (bei Zahlen vom Typ float oder double precision) die Zahl der Nachkommastellen.

Das hohe C

Wir untersuchen die Steuerstrukturen von C und beginnen mit den logischen Operatoren. Bei den Schleifenanweisungen finden wir bemerkenswerte Parallelen zu PASCAL.

Die meisten Vergleichsoperatoren von C entsprechen denen anderer Sprachen: <, <=, >, >=. Beachten Sie, daß das Symbol für gleich == ist und für ungleich !=. Die logischen Operatoren sind && für AND und || für OR. Auch die hierarchische Abfolge der Operatoren entspricht der sonst üblichen Übereinkunft. Komplizierte logische Ausdrücke lassen sich daher ähnlich wie in BASIC anlegen. Die Werte von Zuordnungen kann man in logischen Bedingungen testen. So ist beispielsweise über die Standardfunktion getchar(), die das nächste Zeichen aus dem Eingabekanal holt, eine Bedingung aufzubauen:

```
char count(max line length-1 && (c=
   getchar())) !=/n' && c != EOF
```

Dabei wird zunächst die maximale Zeichenzahl überprüft, dann (falls die Zahl nicht erreicht ist) das nächste Zeichen geholt und schließlich getestet, ob das Zeilen- oder Dateiende vorliegt – alles in einem einzigen logischen Ausdruck.

Auch logische Ausdrücke können Werte haben: FALSE ist NULL und TRUE ist Eins (jeder Wert, der nicht Null ist, wird als TRUE interpretiert). Damit lassen sich logische Ausdrücke auch für Berechnungen einsetzen. Einfache numerische Werte können ohne Operator getestet werden. Eine Ganzzahl ungleich Null verwandelt der Negationsoperator ! in Null, eine Ganzzahl gleich Null in Eins.

```
if(int wert)
```

entspricht

```
if(int wert==0)
```

Außer den normalen logischen Operatoren besitzt C eine Reihe von Bitoperatoren, die mit jedem Typ von integer oder char Bitmanipulationen vornehmen können, die im allgemeinen nur in Assembler zur Verfügung stehen:

&	bitweise AND
|	bitweise inklusiv OR
^	bitweise exklusiv OR
<<n	verschiebt die Ganzzahl n um n Bits nach links
>>n	verschiebt die Ganzzahl n um n Bits nach rechts
~	Einerkomplement

Der &-Operator wird oft für die Ausmaskierung bestimmer Bits eingesetzt, damit ein oder mehrere Bits des Wortes getestet werden können. Der folgende Befehl prüft, ob das Bit 3 eines bestimmten Bytes auf 1 steht:

```
if(c&0x08)
```

Beachten Sie, daß die Schreibweise 0x vor Hexadezimalkonstanten steht und Zahlen, die mit Null beginnen, vom Computer als Oktalzahl verarbeitet werden.

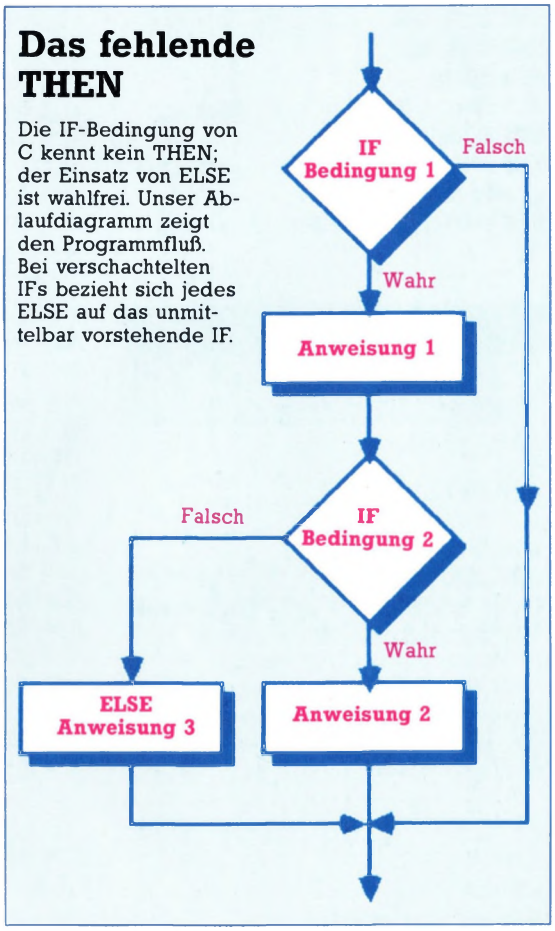
Verzweigungen lassen sich in C mit den üblichen IF-Abfragen ausführen:

```
if(ausdruck)
   anweisung 1
else
   anweisung 2
```

Dabei wird normalerweise ein logischer Aus-

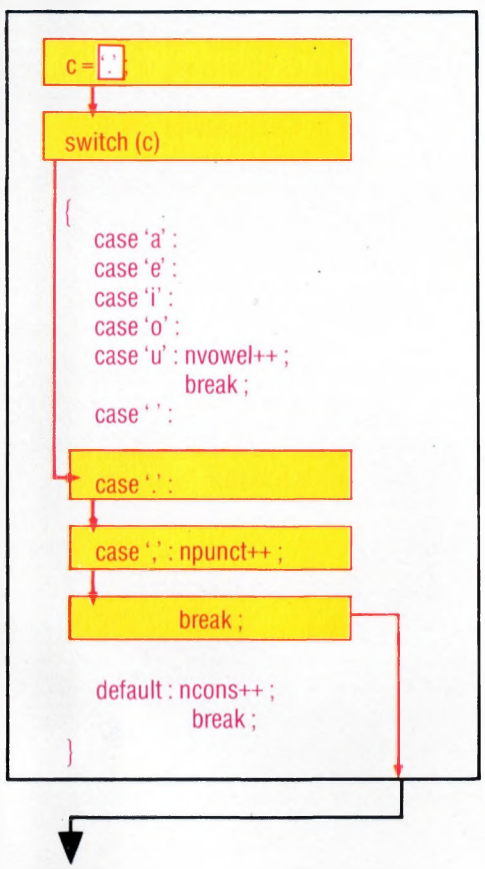
Das fehlende THEN

Die IF-Bedingung von C kennt kein THEN; der Einsatz von ELSE ist wahlfrei. Unser Ablaufdiagramm zeigt den Programmfluß. Bei verschachtelten IFs bezieht sich jedes ELSE auf das unmittelbar vorstehende IF.



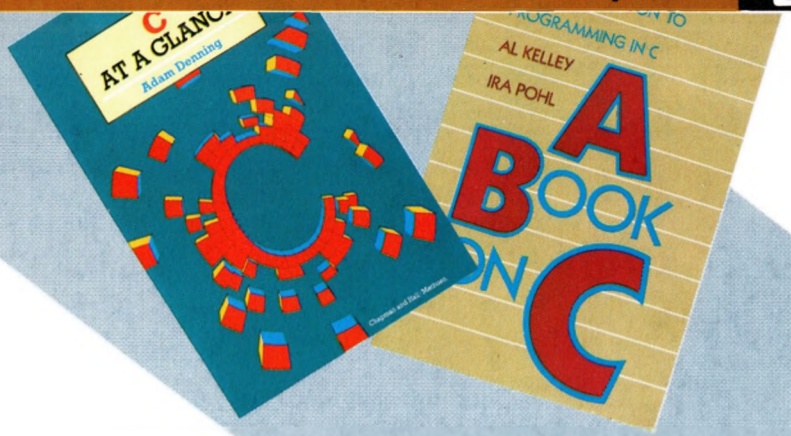
druck verwandt, doch ist jeder Ausdruck möglich, der eine Ganzzahl liefert. Beachten Sie, daß kein then aufgeführt ist. Die Verwendung des else-Teils ist wahlfrei.

Anweisungen können einfaches Format haben, zusammengesetzte Anweisungen müssen in Klammern {} eingeschlossen werden. Wie in vielen anderen Sprachen können verschachtelte ifs Probleme verursachen. Hier gilt aber ebenfalls, daß jedes else dem unmittelbar davor auftretenden if zugeordnet wird. Die Regel läßt sich jedoch ganz einfach mit Klammern umgehen.



Steuerung mit SWITCH

Die SWITCH-Anweisung funktioniert ähnlich wie ON...GOTO in BASIC. Der Parameter von SWITCH wird bewertet und die Steuerung an das entsprechende CASE übergeben. In unserem Beispiel wird je nach dem Wert der Variablen c die entsprechende Zählervariable (nvokal, nsatz, oder nkons) um Eins erhöht und so die Anzahl der Vokale, Konsonanten oder Satzzeichen gezählt. Die Variable c, normalerweise ein Eingabewert, wurde hier mit dem Wert „." geladen, um den Programmablauf zu verdeutlichen. Beachten Sie, daß das „Weiterreichen" von einem CASE zum nächsten ein bequemer Weg ist, mehrere CASE in einem Durchgang zu behandeln.



Wahlmöglichkeiten mit mehr als zwei Alternativen werden von der Anweisung switch gesteuert, die der PASCAL-Funktion case entspricht.

```
switch(ganzzahlausdruck)
{
case value 1:anweisung 1;
case value 2:anweisung 2;
...
default: default anweisung;
}
```

Die Werte von case müssen unterschiedliche Konstante sein. Die Angabe von default ist wahlfrei. Da die Case-Werte Labels sind, springt die Programmausführung nicht automatisch auf die Endanweisung, wenn die Ausführung eines Case beendet ist, sondern spricht das nächste Case an.

break kann das jedoch verhindern. Die Steuerung wird dann an die Anweisung übergeben, die auf switch folgt. (siehe Kasten).

Es gibt drei Arten von Schleifen. Am häufigsten ist die while-Anweisung:

```
while(bedingung) anweisung;
```

Zusammengesetzte Anweisungen werden mit Klammern umschlossen.

Da Bedingungen beliebige Ganzzahlausdrücke sein können, ist es eine weit verbreitete (wenn auch keine gute) Programmierpraxis, den Schleifeninhalt soweit wie möglich als Bedingung anzulegen. Die folgenden Beispiele zeigen zwei Wege, eine bestimmte Schleife zu schreiben. Dabei werden Ziffern, die als alphanumerische Zeichen eingegeben werden, in Ganzzahlen umgewandelt. Das erste nichtnumerische Zeichen beendet schließlich die Routine.

```
int num;
char c;
num=0;
c=getchar();
while(c)='0' && c(='9')
{
  num=num*10+c-'0';
  c=getchar();
{
```

C – Auf einen Blick: Dieses Buch bietet einen tieferen Einblick in C, als der Titel vermuten läßt. BASIC-Kenntnisse werden vorausgesetzt, und C wird so behandelt, wie es auf den meisten Heimcomputern erscheint. (ISBN 0-412-27140-0).

Das ABC von C: „A Book On C" bietet eine ausführliche Behandlung der Sprache und geht auch auf beigeordnete Themen ein – speziell auf das eng mit C verbundene Betriebssystem UNIX. (ISBN 0-8053-6860-4).

Achten Sie darauf, wie die mathematische Berechnung die Zeichen c und '0' einsetzt. Der ASCII-Code wird automatisch in den Typ int umgewandelt, damit eine Rechnung überhaupt stattfinden kann. Die zweite Methode verlegt die Zuordnung zu c in die Schleifenbedingung:

```
num=0;
while((c=getchar()) >= '0' && c <='9')
  num=num*10+c-'0';
```

Da hier nur ein einziger Befehl wiederholt wird, sind keine Klammern nötig.

Die for-Anweisung ist die zweite Methode, Schleifen anzulegen. Sie ähnelt dem FOR. . .NEXT von BASIC und enthält eine Initialisierung, eine Endanweisung (oder Fortsetzung) und eine Schrittanweisung, die bei jedem Schleifendurchlauf aktiviert wird. In den meisten Sprachen wird die Initialisierung mit dem Wert einer Ganzzahlvariablen durchgeführt. Die Endbedingung wird von dem letzten Wert bestimmt, den diese Variable erreichen kann. Die Schrittfunktion addiert einfach einen Ganzzahlenwert auf die Variable.

```
for(expl;exp2;exp3)
  anweisung;
```

expl enthält die Initialisierung und läuft vor der ersten Ausführung der Anweisung ab. Solange die Bedingung wahr (nicht-Null) ist, werden die Fortsetzungsbedingung exp2 und die Schleife wiederholt. exp3 wird am Ende jeder Anweisung ausgeführt. Die BASIC-Schleife:

```
FOR I=1 TO N
  Anweisung(en)
NEXT I
```

entspricht in C:

```
for(i=1;i<=n;i++)
  anweisung;
```

Unser Beispielprogramm zählt und druckt alle Primzahlen zwischen 2 und 1000. Interessant ist besonders der kompakte Code, der durch die Vorinkrementierung mit ++ und das Format der FOR-Schleife erzeugt wird.

Primzahlen

```
/* Das Programm zeigt alle Primzahlen unter
1000 */
main()
{
   int prime count=0, limit=1000, divisor,
   number to test;
/* Beachten Sie die Initialisierung der Variablen
prime count und limit in der Deklarationsanweisung */
   for (number to test=2, number to test<
   limit, ++ number to test)
/* Es folgt die Hauptschleife, die alle Zahlen
zwischen 2 und 1000 testet: */
   {
   divisor=1;
/* Mit %(mod)Operator Divisoren finden */
   while (number to test %++divisor !=0);
/* Die Inkrementierung wird vor dem Test ausgeführt */
/* Die Zahl ist Primzahl, wenn der Divisor den
Wert von number to test erreicht */
   if (divisor==number to test)
/* 1 zum Zähler addieren */
   {
   ++print count;
/* Zehn Primzahlen auf einer Zeile anzeigen */
   printf("%6d",number to test);
   if (prime count % 10==0)
   printf("/n";
   }
}
printf("/n/nes gibt %d Primzahlen unter %d/n",
   prime count,limit);
}
```

Flexible Sache

Um zeigen zu können, wie flexibel die for-Anweisung ist (und C überhaupt), codieren wir die WHILE-Schleife für die Umwandlung eines Eingabestrings in Ganzzahlen mit for:

```
for(num=0;(c=getchar()) >='0' && c<='9';
   num=num*10+c-'0');
```

Beachten Sie, daß der gesamte Schleifeninhalt im Inneren der Bedingung liegt. Obwohl dadurch kompakter und schneller Code entsteht, ist er doch recht unübersichtlich.

C kann Schleifen noch auf eine dritte Art bilden. Sie entspricht dem repeat. . .until von PASCAL. Hier wird der Test im Gegensatz zu den anderen beiden Abläufen erst am Ende der Anweisung durchgeführt:

```
do
  anweisung
while(bedingung);
```

Zwei Zusatzanweisungen gestalten die Schleifensteuerung einfacher als in anderen Sprachen. Wenn die Programmausführung inmitten einer Schleife ein break findet, verzweigt sie sofort auf den unmittelbar hinter dem Schleifenende liegenden Befehl. Das folgende Programmbeispiel liest eine Zeichenfolge ein, die mit einem Return beendet wird, springt aber auch bei der Eingabe von Control C (ASCII-Code 3) aus der Schleife:

```
while((c=getchar())!='/n')
{
  if c==3
    break
  else
    continue /* c weiter bearbeiten */;
}
```

Die Anweisung funktioniert ähnlich, aktiviert aber die nächste Schleife automatisch.

Obwohl mit break und continue alle GOTO-Befehle überflüssig wurden, gibt es in C ein goto. Es wird mit einem Mechanismus der Namensgebung eingesetzt und sollte nur in Spezialfällen verwandt werden.

C in Funktion

Die Programmiersprache C arbeitet hauptsächlich mit Funktionen, die selbst definiert oder vom System vorgegeben werden. Wir untersuchen die Struktur und die Rolle der Funktionen anhand eines Programms.

Die Sprache C baut zwar grundlegend auf dem Konzept der Funktion auf, ist aber dennoch keine Funktionssprache, da die Funktionen als Prozeduren definiert sind.

Mit „Funktion" ist hier ein Vorgang gemeint, dem bestimmte Werte (Argumente oder Parameter) übergeben werden und der nach deren Verarbeitung einen einzelnen Wert zurückliefert. Die übergebenen Werte brauchen keine Grundtypen wie Ganzzahlen oder reelle Zahlen zu sein, sondern können auch strukturierte Typen wie Arrays darstellen.

Auch Blöcke spielen eine große Rolle. In C ist ein „Block" ein selbständiger, von Klammern ({}) umschlossener Codeabschnitt. (Zum Vergleich: PASCAL zeigt Blöcke durch begin . . . end an). Wichtig ist auch der „Bereich" einer Variablen, der Programmabschnitt also, in dem die Variable eingesetzt wird. Da nur wenige BASIC-Versionen mit lokalen Variablen arbeiten, werden viele BASIC-Programmierer sich erst daran gewöhnen müssen, daß der Einsatz von lokalen Variablen auf bestimmte Programmbereiche beschränkt ist. In C können Variablen an jeder beliebigen Stelle deklariert werden. Sie müssen daher genau wissen, von wo Sie sich einfach darauf beziehen können.

Hier die Grundregeln für die Deklaration von C-Variablen:

- Wird eine Variable am Anfang einer Funktionsdefinition deklariert, dann gilt sie für den gesamten Bereich der Funktionsdefinition, also auch für jede andere Funktion, die im Inneren der Hauptfunktion deklariert ist. Da das Programm selbst aus der Funktion main() besteht, gelten am Anfang deklarierte Variablen im gesamten Programmbereich – sie sind „global".

Gut aufgebaute Programme enthalten soweit wie möglich völlig eigenständige Module, die nur mit lokalen Variablen und formalen Parametern arbeiten. Wenn die Ausführung einer Funktion außer lokalen Variablen noch andere Daten einsetzt, wird das „Nebenwirkung" genannt. So ist beispielsweise der Betrieb eines Ein-/Ausgabegerätes eine Nebenwirkung – wie auch der Einsatz einer globalen Variablen. Einige Nebenwirkungen sind positiv – das heißt, sie haben keine Auswirkungen auf die Ausführung anderer Programmteile. Negative Nebenwirkungen stören jedoch den Programmablauf und sollten vermieden werden.

- Eine im Inneren eines gesamten Blockes deklarierte Variable hat nur für den Rest des Blockes Gültigkeit.

Die Variablendeklaration hat besondere Bedeutung, da es Teil der „Philosophie" von C ist, Programme aus Modulen aufzubauen. Jedes dieser Module wird von einer eigenen Datei

dargestellt. Die Regeln, mit denen die Funktionen anderer Dateien in den Bereich einer Variablen eingeschlossen werden können, sind jedoch sehr kompliziert.

Es gibt zwei Möglichkeiten, Argumente oder Parameter an Funktionen zu übergeben:

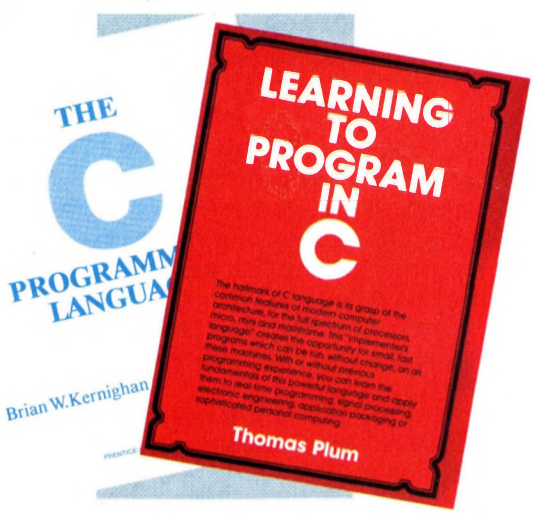
- **Wert:** Durch Anlegen einer neuen Variablen. Variablen dieser Art haben für die Funktion lokale Bedeutung und werden mit den von der aufrufenden Routine übergebenen Werten initialisiert.
- **Bezug:** Hier wird die Adresse der Variablen übergeben, die den Parameter enthält. Auf diese Weise kann die gleiche Variable von der Funktion und der aufrufenden Routine verwandt werden.

C übergibt alle Parameter als Wert. Wir werden jedoch später sehen, daß die Sprache breitgefächerte Möglichkeiten hat, Adressen und Pointer zu verarbeiten und damit auch die Referenzmethode einsetzen kann.

Der formale Aufbau einer Funktionsdefinition lautet:

```
funktion_typ funktion_name( formale_argu-
   ment_liste);
variablen_deklarationen;

   inhalt der funktion;
}
```



Für die wachsende Beliebtheit von C ist auch die Firma Prentice-Hall verantwortlich, die umfassendes Lesematerial über die Sprache herausgab, darunter die erste Sprachdefinition von Kernighan und Ritchie. Der hohe Preis dieses dünnen Buches mag vielleicht abschrecken, doch bleibt es bis zur Definition eines internationalen Standards die Grundlagenbeschreibung der Sprache. Ein weiteres wichtiges von Prentice-Hall herausgegebenes Buch, „Learning to Program in C", liegt als deutsche Übersetzung vor: „Das C Lernbuch" Verlage Carl Hanser und Prentice-Hall, ISBN 3-446-14165-0.

Die Angabe **funktion_typ** kann weggelassen werden, wenn der zurückgegebene Wert vom Typ **int** ist. Die **formale_argument_liste** enthält eine Liste der Variablen, in die die Werte beim Einsatz der Funktion gespeichert werden.

Die entsprechende Funktion wird folgendermaßen aufgerufen:

```
funktion_name (aktuelle_argument_liste)
```

Der Aufruf kann an jedem Punkt erfolgen, an dem ein Wert des von der Funktion gelieferten Typs eingesetzt wird. Wenn der zurückgegebene Wert nicht vom Typ **int** ist, muß der Funktionsname vor dem Aufruf als Variable des entsprechenden Typs deklariert werden. Die aktuelle Argumentliste enthält alle Werte (oder Variablen mit diesen Werten), die an die formalen Argumente der Funktion übergeben werden. Die aktuellen und formalen Argumente müssen natürlich in beiden Listen vom Typ und von der Position her übereinstimmen.

Mit **return** übergibt eine Funktion den gewünschten Wert an die aufrufende Routine. Es ist eine beliebige Zahl dieser Befehle möglich, doch sollte vor Verlassen der Funktion mindestens einer ausgeführt werden.

Das folgende Programm liest eine Reihe von reellen Zahlen ein und zeigt die größte und die kleinste an. Das Programm enthält drei Funktionen. **ausgabekopf** hat keine Argumente und auch keinen **return**-Befehl. Das bedeutet, daß der von der Funktion zurückgegebene Wert nicht definiert ist. Da die aufrufende Routine ihn nicht benötigt, wird er einfach „weggeworfen". Die Funktion hat Nebenwirkungen, die jedoch alle positiv sind.

Werte für Variablen

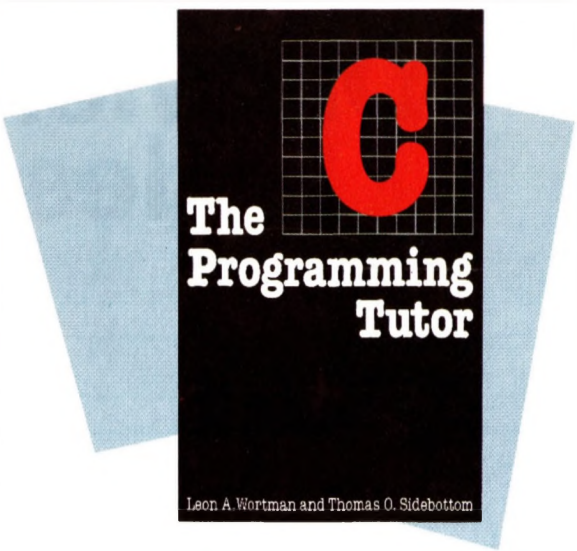
Das Programm arbeitet weiterhin mit der Standardfunktion **scanf**, die Eingaben auf ähnliche Weise formatiert wie **printf** die Ausgaben. Der erste Parameter der Funktion ist ein String mit den Formatinformationen der Eingabewerte. Die weiteren Argumente von **scanf** sind die Variablen, in die die Werte eingelesen werden. Da Parameter sich jedoch nur als Werte übergeben lassen, können sie an die aufrufende Routine auch keine Werte zurückgeben. Die Parameter müssen daher statt der Variablen selbst die Adressen der Variablen enthalten. Vor jeder beim Aufruf von **scanf** angegebenen Variable muß daher der Adreßoperator & stehen.

```
#include<stdio.h>
main ()
{
   int count, size_of_input,
   double smallest, largest, number, min(),
      max();
/* beachten sie die deklaration der funktionen
   min and max */
/* beachten sie weiterhin, daß im inneren
   eines blockes deklarierte variablen außer-
   halb des blockes nicht verfuegbar sind */
   printheading();
/* nach aufruf der funktion wird der zurueckge-
   gebene wert als int angesehen und 'wegge-
   worfen', d. h. nirgendwo anders eingesetzt
      */
   printf ("/nsize of input:");
   scanf ("%d",&size_of_input);
/* herausfinden, wieviele zahlen eingelesen
   werden sollen */
   printf ("Jetzt %d reelle Zahlen eingeben :/n",
      size_of_input);
/* die erste zahl holen, die im augenblick die
   kleinste und groesste ist */
   scanf ("%if", &number);
   largest=smallest=number;
/* jetzt die anderen zahlen holen */
```

```
    for (count=2; count<=size_of_input;
    ++count)
    {
       scanf("%if", &number);
       smallest = min (smallest, number);
       largest=max (largest, number);
    }
    printf ("/nsmallest is %f/nlargest is
    %f/n", smallest, largest);
}
/* hier die funktionsdefinitionen */
printheading(,
/* da kein typ gebraucht wird, int annehmen */
{
    printf ("/n%s/n%s/n%s/n",
       "Zuerst die Groesse des Zahlensets ein-
          geben",
       "dann entsprechend viele reelle Zahlen
          eingeben.",
       „Die groesste und kleinste Zahl werden
          dargestellt");
}
double min(x,y);
double x,y;
/* beachten sie die parameterdeklaration */
{
    if (x<y)
       return (x);
    else
       return(y);
}
double max (x,y);
double x,y;
{
    if (x>y)
       return(x);
    else
       return(y);
```

Viele Compiler verstehen „Compileranweisungen", die in den Programmtext eingebaut sind. Bei C wurde dieses Konzept weiter ausgebaut als bei den meisten anderen Sprachen. Jeder C-Compiler enthält daher einen „Vorprozessor", der bestimmte „Steuerzeilen" als Befehle erkennt und das Programm verändert, bevor es überhaupt dem Compiler übergeben wird. Sie sind am vorangehenden # auszumachen.

Die Vorprozessoranweisung **#include<dateiname>** oder **#include"dateiname"** befiehlt dem Vorprozessor, an dieser Stelle den Inhalt der angegebenen Datei einzuschließen. Einige Systeme kennen keinen Unterschied zwischen spitzen Klammern <> und Anführungszeichen "", andere suchen dadurch jedoch an unterschiedlichen Plätzen nach den angegebenen Dateien. Spitze Klammern werden normalerweise für Systemdateien eingesetzt und Anführungszeichen für selbstdefinierte Dateien. Die Funktionen **printf** und **scanf** sind in der Systemdatei **stdio.h** enthalten. Alle Programme, die damit arbeiten, müssen daher die Anweisung **#include<stdio.h>** enthalten.



Das von Prentice-Hall herausgegebene Buch „The C Programming Tutor" ist eine ausgezeichnete Einführung in die Sprache. Klar und deutlich geschrieben, werden dem Leser komplizierte Themen nahegebracht. Das Buch ist besonders für Leser geeignet, die außer BASIC keine anderen Programmiersprachen kennen (ISBN 0-13-110024-6).

Da diese Aufrufe geschachtelt werden dürfen, können auch eingeschlossene Dateien mit **#include** weitere Dateien einbeziehen.

Die Anweisung **#define** macht den Einsatz von Macros möglich. Die einfachste Form ist dabei die Definition einer Konstanten. **#define EOF–1** ersetzt beispielsweise jedes Auftreten des Namens EOF durch –1.

```
#define PI 3.14159
#define EQ ==
```

Derart definierte Namen werden in Großbuchstaben geschrieben und normale Variablen in Kleinbuchstaben.

Auch Parameter können in Macrodefinitionen wie

```
#define SUMSQ(x,y)((x*x)+(y*y))
```

vorkommen, so daß eine Art Funktionsdefinition wie in BASIC vorliegt. Das Auftreten von SUMSQ(3,4) würde dabei durch ((3*3)+ (4*4)) ersetzt werden. Die Klammern sind aus Sicherheitsgründen mit aufgeführt: Das Macro könnte nun in einer Situation eingesetzt werden, in der Klammern nötig sind.

Oft wird auch die Anweisung **#if...#else...#endif** verwandt, die die bedingte Compilierung steuert. Wenn der konstante Ausdruck nach der #if-Anweisung mit wahr (nicht-Null) bewertet wird, werden die darauf folgenden Codezeilen compiliert. Ergibt die Bewertung des Ausdrucks falsch (Null), werden die auf **#else** folgenden Zeilen compiliert. Diese Struktur ist besonders bei der Programmentwicklung praktisch, wenn zusätzliche **printf**-Befehle Klarheit über den Programmablauf bringen sollen. Hier können Anweisungen wie **#define DEBUG 1** eingesetzt werden, die bei Bedarf mit

```
#if DEBUG
   printf(werte . . .)
#endif
```

die gewünschten Werte ausdrucken. Nach Beendigung der Fehlersuche muß nur **#define DEBUG 0** angegeben werden, und eine erneute Compilierung schließt diese Anweisungen nicht mehr mit ein.

Klassenbewußt

Wir untersuchen die vier „Speicherklassen" für Variablen der Programmiersprache C und initialisieren Arrays. In C haben Variablen einen exakt definierten Status, je nachdem, wie sie dem Speicher zugeordnet werden.

Außer einer Typenzuordnung haben alle C-Variablen auch eine Speicherklasse, die bestimmt, wie der Compiler sie verarbeitet und wie der Speicher für sie reserviert wird. Es gibt vier Klassen: automatic, extern, register und static. Die Klassenzuweisung geschieht ebenfalls mit diesen Schlüsselwörtern, die in der Variablendeklaration vor der Typenbezeichnung stehen:

```
extern double x, y;
```

Automatic ist die Standardklasse fast aller Variablen. Jede Variable, die innerhalb einer Funktion definiert wird, nimmt die Klasse automatic an und hat nur lokale Bedeutung. Bei jedem Aufruf der Funktion wird dieser Variablen dann neuer Speicherplatz zugewiesen, der beim Verlassen der Funktion wieder frei ist. Es ist nicht möglich, Variablen dieser Art zwischen zwei Aufrufen einer Funktion beizubehalten. Das gleiche trifft auch auf alle Variablen zu, die im Inneren eines Codeblocks definiert wurden und in Klammern {} eingeschlossen sind. Da main() ebenfalls eine Funktion ist, werden alle vorkommenden Variablen normalerweise als automatic angelegt.

Externe Variablen sind global und können von jedem Punkt des Programms aus angesprochen werden – unter bestimmten Umständen sogar von Funktionen, die in der Quelldatei der Deklaration nicht enthalten sind. Variablen, die außerhalb eines Funktionsmoduls deklariert werden, sind automatisch extern. Auch externe Variablen lassen sich an jeder beliebigen Stelle einer Funktion oder eines Blocks deklarieren. Sie bleiben nach dessen Ablauf erhalten und stehen nach ihrer Deklaration beliebig zu Verfügung.

Zwei externe Variablen, die in zwei oder mehr Quelldateien den gleichen Namen erhalten, werden beim Verbinden der Dateien als die gleiche Variable angesehen. Eine lokale Variable, die mit dem Namen einer globalen Variablen deklariert wird, „maskiert" diese im lokalen Bereich, so daß sich der Name im Inneren der Funktion (oder des Blocks) auf die lokale Variable bezieht und außerhalb auf die globale Variable.

Register-Variablen verhalten sich wie automatic-Variablen und werden in einigen Fällen auch ebenso behandelt. Die Zahl und Größe der für sie zur Verfügung stehenden Register mit schnellem Zugriff hängen vom Prozessor ab. Registervariablen sollten stets sparsam eingesetzt werden. Deklarieren Sie sie so spät wie möglich und geben Sie sie sofort frei, wenn sie nicht mehr benötigt werden. Registervariablen steuern oft Schleifen.

Static-Variablen sind normalerweise lokal, unterscheiden sich von lokalen Variablen aber dadurch, daß ihr Wert und Speicherplatz zwischen zwei Ausführungen einer Funktion (oder eines Blocks) erhalten bleibt. Statische Variablen können beispielsweise zählen, wie oft Funktionen aufgerufen werden. Auch können sie Daten „verstecken", da ihre Werte nicht wie externe Variablen von außen zugänglich sind. Eine static-Variable wird extern für eine Reihe von Funktionen deklariert und ist für diese Funktionen global verfügbar.

Es geht um Arrays

Arrays werden wie andere Variablen mit ihrer Größe (das heißt der Anzahl ihrer Elemente) deklariert. Die Größe steht hinter ihrem Namen in eckigen Klammern:

```
int intarray[100];
```

Damit wird Speicherplatz für die Arrayelemente intarray[0], intarray[1] etc. bis intarray[99] reserviert. Subscripts fangen immer bei Null an, wobei die Deklaration die Gesamtzahl der Elemente enthält. In diesem Fall gibt es daher kein intarray[100]. Static- oder external-Arrays werden in der Deklaration durch Hinzufügen einer in Klammern eingeschlossenen Werteliste initialisiert:

```
static int tage_im_monat[12] =
  {31,28,31,30,31,30,31,31,30,31,30,31};
```

Wenn die Liste nicht vollständig ist, werden die restlichen Elemente auf Null gesetzt. Ohne Initialisierung stehen alle Elemente eines statischen oder externen Arrays auf Null. Automatic-Arrays können nicht initialisiert werden. Da ihr Speicherplatz anfangs mit beliebigen Werten gefüllt ist, sollten Sie nicht davon ausgehen, daß ihre Elemente auf Null stehen.

Bei der Initialisierung von Arrays muß die Größe nicht angegeben werden, da C dafür au-

tomatisch die Zahl der angegebenen Werte nimmt. Die obenstehende Deklaration könnte auch so aussehen:

```
static int tage_im_monat[] =
  {31,28,31,30,31,30,31,31,30,31,30,31};
```

Besonders bei Strings und Arrays vom Typ char ist das sehr praktisch, da der initialisierte String nur in Anführungsstriche eingeschlossen wird. Beide Deklarationen sind gleichwertig.

```
static char st[] = "hallo";
```

ist gleichwertig mit

```
static char st[] = {'h','a','l','l','o'};
```

Beachten Sie jedoch, daß diese Strings nicht dynamisch sind. Die Länge des Strings kann nicht von der deklarierten Länge abweichen.

C kann Arrays mit fast jeder Zahl von Dimensionen verarbeiten. Bei Arrays mit zwei oder mehr Dimensionen wird jedes Subscript in ein eigenes Paar von eckigen Klammern eingeschlossen. Die Deklaration eines zweidimensionalen Arrays mit vier mal fünf Elementen sieht daher so aus:

```
int zweiarray[4][5];
```

Arrays lassen sich auch als Parameter an Funktionen übergeben, werden dann aber über eine Bezugsadresse angesprochen. Dabei wird die Adresse des ersten Elementes an die Funktion übergeben. Veränderungen, die von der Funktion im Array vorgenommen wurden, bleiben nach Verlassen der Funktion erhalten. Innerhalb einer Funktion muß die entsprechende Größe eines Arrays nicht deklariert werden, da sie bereits bekannt ist.

Unsere Beispielprogramme enthalten viele dieser Konzepte. Wir gehen davon aus, daß ein Zufallszahlengenerator und auch dessen Steuerfunktionen als Quelldateien vorhanden sind und beim Verknüpfen in die Programme eingebunden werden können. Das kurze Testprogramm ruft eine große Zahl von Zufallszahlen auf und prüft über die Zahlenhäufigkeit, ob eine gleichmäßige Verteilung besteht.

Listing 1, gespeichert in file 1

```
# define MULT 25173
# define MODULUS 65536
# define INCREMENT 13849
static int seed;

/* diese Deklaration geschieht extern, so dass die
Variable seed allen Funktionen dieser Datei
for (j = 0; j < NUM_OF_GROUPS; ++j)
{ printf ("%d—%d/n", groups[j]—
10, groups[j], frequencies [j]) }
zur Verfuegung steht, ausserhalb dieser Datei aber
nicht angesprochen werden kann. Der Name „seed"
steht daher in anderen Dateien wieder frei zur Ver-
fuegung */
randomise(s)

int s
{
   seed = s;
}
random (n)
int n;
/* ergibt eine Zufallszahl zwischen 0 and n */
{
   seed = (MULT * seed + INCREMENT) %
   MODULUS;
/* Die lineare Kongruenzmethode */
   return ((int) ((double) seed / (double)
   MODULUS *
   (double) n + 0.5));

/* Beachten Sie, wie die Ganzzahlenwerte mit Spezial
formen in Fliesskommazahlen und wieder in Ganz-
   zahlen gewandelt werden. Die + 0,5 runden auf die
   naechsthoehere Ganzzahl */
}
double rnd()

/* ergibt eine Zufallszahl zwischen 0 and 1 */
{
   seed = (MULT * seed + INCREMENT) %
   MODULUS;
   return ((double) seed / (double) MODULUS);
}
```

Listing 2, gespeichert in file 2

```
# define SEED 17
# define SIZE 100
# define LIMIT 10000
# define NUM_OF_GROUPS 10
int groups [] = {10,20,30,40,50,60,70,80,90,100};

/* externe Array lassen sich initialisieren. Dieses Array
enthaelt die Gruppenbegrenzungen fuer die Zaehl-
frequenzen */

int frequencies [10];
/* dieses externe Array wird mit Nullen initialisiert */
main ()
int r;
{
      randomise (SEED);
      register int i;

/* Durch den Einsatz der Klasse register fuer die
Schleifenvariablen wird die Verarbeitung sehr schnell.
Deklarieren Sie die Variable so spaet wie moeglich */

      for (i = 0; i < LIMIT; ++i)
      {
/* 10000 Zufallszahlen zwischen 0 und 100 holen */
         r = random (SIZE);
         register int j;
/* testen, zu welcher Gruppe sie gehoeren */
         for (j = 0: j < NUM_OF_GROUPS;
            ++j)
            {
            if (r < groups[j])
{
/* Zufallszahl in die entsprechende Gruppe eintragen
und Schleife verlassen */
               ++ frequencies(j);
               break;
            }
         }
/* eine Verteilungtabelle ausgeben, mit der die gleich-
maessige Verteilung geprueft werden kann */
         for (j = 0; j < NUM__OF__GROUPS; ++j)
         { printf ("%d–%d=%d/n", groups[j]–
10, groups[j], frequencies [j]) )}
```

Funktionen zur Stringbearbeitung

strcmp(s1,s2) – s1 und s2 sind Pointer auf char (strings). Zurückgeliefert wird ein Ganzzahlenwert. Bei s1<s2 ist er unter, bei s1=s2 gleich und bei s1>s2 über Null.
strncmp(s1,s2,n) – Wie strcmp, nur werden höchstens n Zeichen verglichen.
strlen(s) – s ist ein Pointer auf char. Die Funktion liefert die Stringlänge als Ganzzahl.
index(s,c) – s ist ein Pointer auf char und c ist char. Liefert Pointer auf erstes Vorkommen von c in s; oder NULL, falls es c in s nicht gibt.
rindex(s,c) – ähnlich wie index, nur wird die Suche vom rechtsaußen stehenden Zeichen aus durchgeführt.
strcat(s1,s2) – Hängt eine Kopie von s2 an das Ende von s1 und liefert s1. Es wird vorausgesetzt, daß Länge von s1 für alle Zeichen ausreicht.
strncat(s1,s2) – Hängt maximal n Zeichen an das Ende von S1 und liefert s1.
strcpy(s1,s2) – Kopiert Inhalt von s2 bis zu dem '/' in s1. Vorausgesetzt wird, daß Länge von s1 ausreicht. Der alte Inhalt von s1 wird gelöscht. Zurück kommt der Wert von s1.
strncpy(s1,s2,n) – Kopiert maximal n Zeichen von s2 in s1 und hängt ein „/' an, außer n ist größer als s2.

Wegweiser

Mit dem Variablentyp „Pointer" lassen sich in der Programmiersprache C absolute Speicheradressen ansprechen. Wir sehen uns Deklaration und Einsatz der Pointer genauer an und untersuchen dann Methoden der Stringverarbeitung.

Da C als Ersatz für Assembler entwickelt wurde, bietet diese Programmiersprache Möglichkeiten, die andere Hochsprachen nicht haben. So kann sich C beispielsweise direkt auf Maschinenadressen beziehen. Zwar kennt auch PASCAL den Variablentyp „Pointer", doch ist der Bezug zu Maschinenadressen dort nicht genau definiert. In C enthalten die Pointertypen direkte Maschinenadressen, die auf einer Multitasking-Maschine jedoch kaum praktischen Nutzen haben.

Pointervariablen können aber Hardware direkt steuern. Jede Variable, egal welchen Typs, besitzt eine Adresse, die einem Pointer zugeordnet werden kann. Speziell bei Strings ist es oft praktischer, mit Pointern auf einzelne Elemente zuzugreifen, statt sie mit Arraysubscripts anzusprechen.

Pointervariablen werden mit dem Zeichen * deklariert.

```
int *p;
```

deklariert die Pointervariable p. Beachten Sie, daß die mit int deklarierten Pointer nur auf einen bestimmten Variablentyp (den „Basistyp") zeigen können. Wenn Sie einen Pointer auf den Variablentyp char setzen wollen, müssen Sie daher

```
char *p;
```

eingeben etc. Da der * hier nur die Bedeutung „einen Pointer anlegen" hat, zeigt der Pointer noch auf keine Adresse.

Mit dem Zeichen & werden Pointer mit bestimmten Werten initialisiert. Steht & vor einer Variablen, liefert sie die Adresse der Variablen und nicht deren Wert.

```
P=&i;
```

initialisiert p als Pointer auf i. Doch auch * kann den Wert anzeigen, auf den eine Pointervariable zeigt.

```
i=*p;
```

speichert den Wert, auf den p zeigt in i,

```
i=p;
```

dagegen die Adresse, auf die p zeigt. Für die Bearbeitung absoluter Adressen brauchen Sie jedoch eine „cast" (Formatierung), um Fehlermeldungen des Compilers zu vermeiden:

```
p=(int*) 1000;
```

Pointervariablen lassen sich auch mit den Operatoren für In- und Dekrementierung (++ und – –) einsetzen. Sie berücksichtigen dabei automatisch den Typ der Variablen, auf die der Pointer zeigt. Bei einem System mit Ganzzahlen im Vier-Byte-Format zeigt p daher auf das erste Element eines Ganzzahlenarrays, p++ inkrementiert die Adresse in p um Vier und zeigt so auf das nächste Element. Das gleiche würde passieren, wenn wir p=p+1 verwenden: Hier wird bei der Inkrementierung die Größe des angesprochenen Elementes berücksichtigt und nicht nur Eins addiert. Damit wird klar, daß Pointer nicht mit Ganzzahlen identisch sind, selbst wenn sie Ganzzahlen enthalten.

Pointer können bei Funktionsaufrufen als Parameter übergeben werden. Dies schafft die Möglichkeit, Werte oder Bezug (reference) anzusprechen – das heißt, der Wert des Pointers wird innerhalb der Funktion zwar an eine lokale Variable übergeben, doch zeigt der Poin-

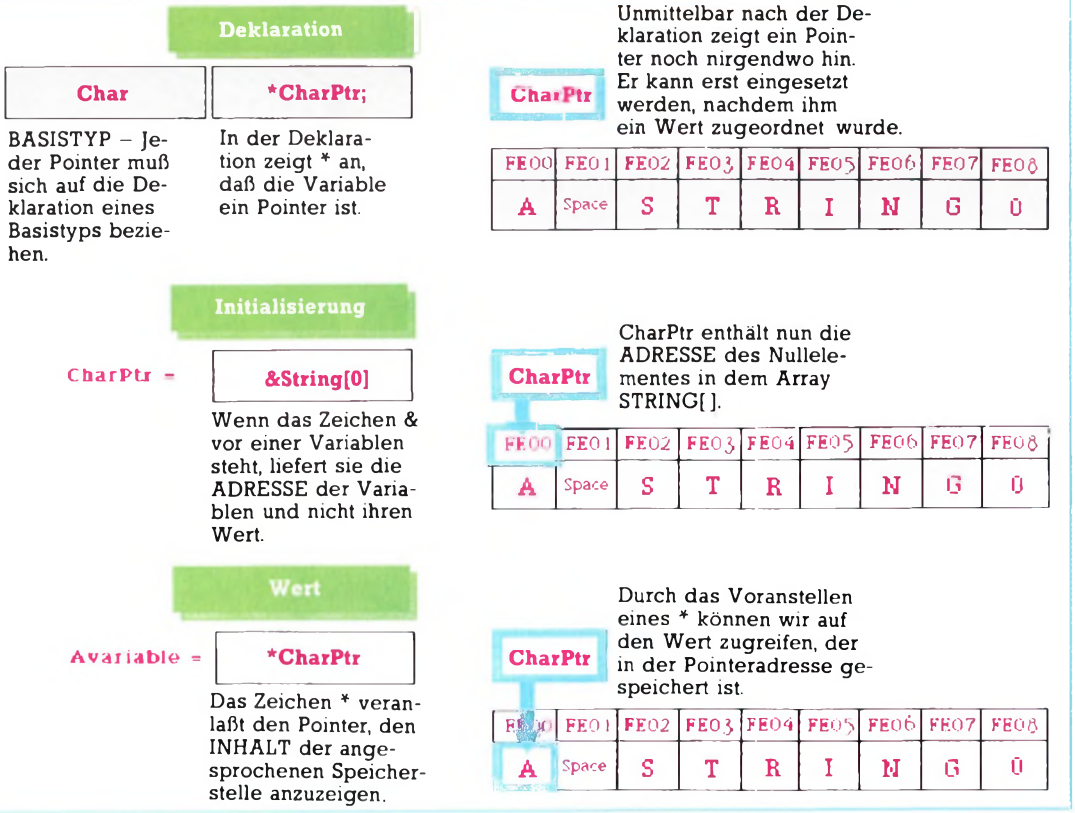
Der Bubblesort in C

```
bubble(a,n)
int a[],n,putinorder();
/* a ist ein Array mit n Ganzzahlen, die sortiert
   werden sollen */
{
int i,j;
   for (i=o;i<n—1;++i)
      for (j=n—1;i<j;— —j)
         putinorder(&a[j—1],&a[j]);
/* Die Adressen der Arrayelemente als Para-
   meter uebergeben */
}
putinorder(x,y)
int *x,*y,swap();
/* x und y sind Ganzzahlenpointer */
{
   if(*x>*y)
      swap(x,y);
}
swap(x,y)
int *x,*y;
{
   int temp;
   temp = *p;
   *p=*q;
   *q=temp;
/* beachten Sie, daß die Adressen der beiden
   Ganzzahlen zwar als Wert uebergeben wer-
   den, die Funktion sich darueber aber auf die
   Variablen beziehen kann, die sonst ausser-
   halb ihres Bereiches liegen wuerden */
{
```

Zahlenpointer
Unser Programm zeigt, welch enge Beziehung in C zwischen Pointern und Arrays bestehen. Der traditionelle „Bubble Sort Algorithmus" arbeitet mit einem Ganzzahlenarray. C-Pointer sind für den Zugriff auf Arrayelemente oft besser geeignet als Subscripts.

Auf den Punkt

Das Bild zeigt drei Stadien des Pointers CharPtr beim Ansprechen unterschiedlicher Elemente des Stringarrays STRING[]. Beachten Sie, daß CharPtr nach seiner Initialisierung eine Adresse zurückliefert, *CharPtr jedoch den in dieser Adresse gespeicherten Wert. Pointer lassen sich in- und dekrementieren und können so auf verschiedene Elemente des Arrays zeigen. Der Pointerwert wird dabei auf den in der Deklaration angegebenen Basisdatentyp eingestellt. So veranlaßt der Basistyp char Schritte im Ein-Byte-Rhythmus, während int auf den meisten Systemen Zwei-Byte-Schritte auslöst.



ter immer noch auf das gleiche Datenelement, wie in dem aufrufenden Programm. Änderungen des gespeicherten Wertes bleiben auch nach dem Rücksprung erhalten.

Zwischen Pointern und Arrays besteht eine enge Beziehung. Bei der Deklaration eines Arrays ist der Arrayname (ohne Subscript) ein Pointer auf das erste Arrayelement. Bei int a[100] *p ist daher p=a; das gleiche wie p=&a[0]. Der Zugriff auf Elemente eines Arrays oder Subarrays läßt sich mit Pointern oft schneller ausführen als mit Subscripts.

Strings sind eindimensionale Arrays von char. Da sie die Grundlage vieler Anwendungen bilden, wurde C mit Spezialmöglichkeiten und Standardfunktionen ausgerüstet, die die Stringbearbeitung sehr erleichtern. So steht in C hinter einem Array von char (mit einem String) immer das Zeichen \0 (ASCII-Code für Null). Da dies an jeder Arrayposition auftreten kann, entsteht selbst bei den auf feste Längen programmierten Arrays der Eindruck dynamischer Stringverwaltung. Bedenken Sie jedoch, daß auch das Zeichen \0 eine Stelle des Arrays belegt. Die Arraylänge muß daher um Eins höher sein als die maximal vorgesehene Zeichenzahl.

In doppelte Anführungszeichen eingeschlossene Stringkonstanten können jedem Array von char mit ausreichender Länge zugeordnet werden. \0 wird automatisch ergänzt. Die Variablenzuordnung geschieht über das Array selbst oder mit einem Pointer auf char.

```
char *s;
*s="abc";
```

veranlaßt, daß die vier Zeichen a, b, c und \0 in vier aufeinanderfolgenden Speicherstellen abgelegt werden, die bei der in s gespeicherten Adresse anfangen (S zeigt auf das Zeichen a). Nach ++s zeigt s auf b etc.

Pointer sind normale Variablen und belegen ebenfalls eine Speicherstelle mit Adresse. Es ist daher möglich, Pointer auf Pointer zeigen zu lassen. Die Programmierung kann sich dadurch zwar schwieriger gestalten, doch gibt es einige Bereiche, in denen Pointerarrays praktisch sind.

Es gibt zwei Spezialparameter, die an die Hauptfunktion übergeben werden können und Zugang zu den Befehlsparametern geben: argc liefert die Zahl der in der Befehlszeile enthaltenen Argumente (durch Leerzeichen voneinander getrennt) und argv ein Array von Pointern auf die Strings mit argc-Elementen. Jedes Element in argv zeigt auf das erste Zeichen dieses Befehlsstrings. Das folgende Programmbeispiel verdeutlicht diesen Vorgang es stellt in diesem Fall die Befehlsargumente zeilenweise dar:

```
main(argc,argv)
int argc;
char *argv[];
/* Deklaration erzeugt Pointerarray */
{
   int i;
   for (i=1;i<argc;i++)
      printf("%s/n",argv[i]);
/* Beachten Sie, daß das % Format einen
Pointer auf einen String erfordert */
}
```

Bewegliche Typen

Über die anwenderdefinierten Datentypen von C läßt sich diese sehr flexible Sprache noch weiter verbessern und erweitern. In dieser Folge untersuchen wir einige der Erweiterungen und sehen uns ihr Zusammenspiel in einem Bridge-Programm an.

In der auf Flexibilität ausgelegten C-Sprache lassen sich selbstverständlich auch die Standardfunktionen erweitern. Wir haben bereits gesehen, wie anwenderdefinierte Funktionen in eigenen Dateien gespeichert und mit include in ein Programm integriert werden. Auch normale Datentypen wie int, char etc. lassen sich verbessern. typdef kann die Standardtypen beispielsweise mit neuen Namen versehen. Hier einige Beispiele:

```
typdef int meter;
typdef double vektor [10];
typdef char *string;
```

Die Befehle ermöglichen, im Programm Variablen zu deklarieren.

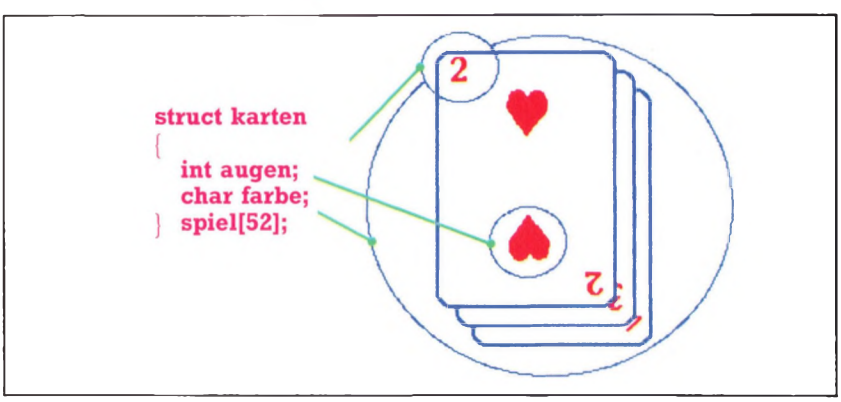
Dies scheint nicht allzuviel Sinn zu machen, hat aber zwei ganz bestimmte Gründe: Die Programme werden dadurch übersichtlicher und verursachen bei der Übertragung von einem Maschinentyp zum anderen weniger Probleme. Normalerweise ist die Zahl der Bytes einer Ganzzahl nicht standardisiert. Ein Programm mit Ganzzahlen im Vier-Byte-Format kann nur mit viel Mühe auf eine Maschine übertragen werden, die mit dem Zwei-Byte-Format arbeitet.

Ein typdef am Programmanfang läßt sich viel leichter ändern, als jedes einzelne Vorkommen dieser Definition während des gesamten Programms. typdefs stehen in einer include-Datei.

Wie PASCAL kann auch C Datenelemente unterschiedlicher Typen in Strukturen zusammenfassen. Der Befehl struct ähnelt dabei dem RECORD von PASCAL. Hier ein Beispiel:

```
struct karten
{
   int augen;
   char farbe;
}
```

Mit dem Typ struct kann der C-Programmierer unterschiedliche Datentypen zu einer Struktur zusammenstellen. Ein Kartenspiel kann beispielsweise mit Farbe und Augenzahl dargestellt und in einer Struktur zusammengefaßt werden.



Hier wird eine Struktur definiert, die aus zwei Teilen besteht – einer Ganzzahl und einer Zeichenfolge – und zusammen eine Spielkarte bezeichnet. Variablen dieses Typs können von dieser Programmposition an deklariert werden.

```
struct karten spiel [52];
```

legt ein Kartenspiel an. Die Variable läßt sich aber auch mit einer einfachen Strukturdefinition deklarieren:

```
struct karten
{
   int augen;
   char farbe;
} spiel [52];
```

Die Bezeichnung (hier karten) kann auch weggelassen werden, wenn alle Variablen an einer Stelle deklariert werden,

```
struct
{
   int augen;
   char farbe;
} spiel [52], blatt [4][13], * gespielt;
```

doch ist dies schlechter Programmierstil.

Pointervariable möglich

Strukturen lassen sich als Ganzes zuordnen, als Argumente an eine Funktion übergeben, oder wie bei einer Standardfunktion als Werte zurückgeben:

```
karten [1][5] = spiel [27];
```

Die einzelnen Elemente können aber auch mit einem Punkt als normale Variable angesprochen werden, oder mit → als Pointervariable:

```
gespielt → augen = spiel [34]. augen;
```

Die Elemente verhalten sich wie normale Variablen des entsprechenden Typs. Die Strukturen werden wie Arrays initialisiert, indem hinter den Variablennamen in Klammern eine Elementenliste aufgeführt wird.

Zur Speicherung ihrer Elemente reservieren die Strukturen aufeinanderfolgende Computerworte. „unions" verwenden jedoch den gleichen Platz für alle Elemente und machen so eine andersartige Verwendung des Arbeitsspeichers möglich.

Wenn eine Maschine beispielsweise vier Bytes für eine Ganzzahl braucht, und diese Bytes einzeln angesprochen werden sollen, kann ein union die gleichen vier Bytes für ein int oder vier chars einsetzen. Das sieht dann folgendermaßen aus:

```
union int_or_char
{   int int_teil;
    char char_teil [4];
}
```

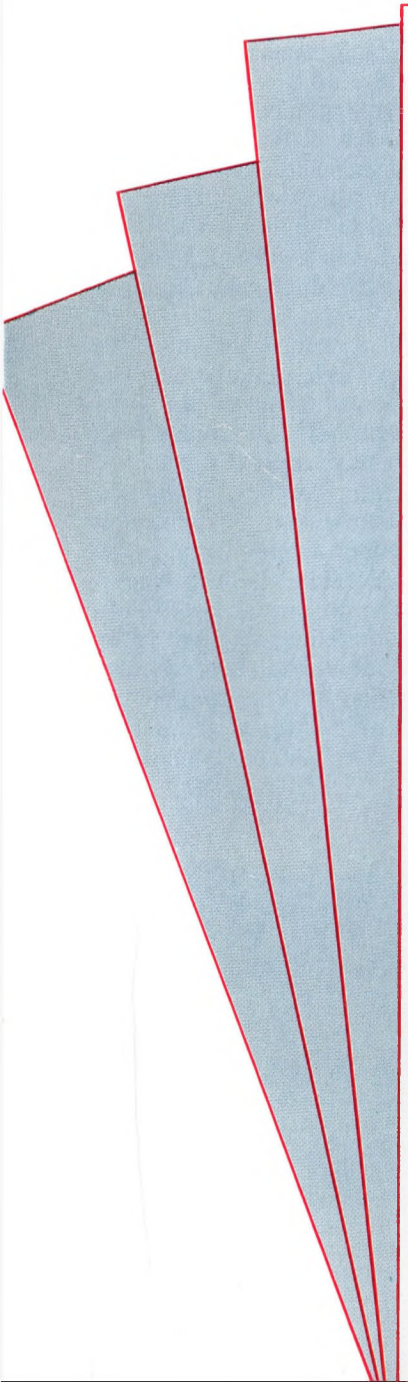
Die Elemente von union werden wie die einer Struktur angesprochen.

Der normale Sprachgebrauch bezeichnet die Elemente einer Struktur als Felder. In C bezieht sich dieser Begriff jedoch auf einen Elemententyp, der auch einzelne Bits ansprechen kann. Wir haben bereits gesehen, daß C Bitoperatoren besitzt. Mit Feldern können Sie nun jede Bitgruppe eines Wortes mit einem oder mehr Bits über einen Namen ansprechen.

Auf einer Maschine mit 16-Bit-Worten und einem int-Typ mit 16 Bits ist folgende Definition möglich:

```
struct wort_in_bytes
{
    unsigned byte0 : 8, bytel : 8;
} struct wort_in_bits
{ unsigned bit0 : 1,bit 1 : 1,bit 2 : 1,bit 3 : 1,
  bit4 : 1,bit5 : 1, bit6 : 1,bit7 : 1,bit8 : 1,
  bit9 : 1,bit10 : 1,bit11 : 1,bit12 : 1,bit13 : 1,
  bit14 : 1,bit15 : 1;
}
union wort
{ int w;
  struct wort_in_bytes w8;
  struct wort_in_bits w1;
}
```

Sie können nun das Speicherwort als Ganzes ansprechen – als zwei Bytes oder als 16 Bits. Beachten Sie den Datentyp unsigned, der int entspricht, aber nur positive Werte aufnehmen kann. Bei einer Größenfestlegung dieser Art müssen Sie jedoch darauf achten, daß nur geeignete Werte zugeordnet werden. Ein Element mit Bitgröße kann beispielsweise nur die Werte 0 bis 1 aufnehmen.



Ein Bridgespiel

Das folgende Programmbeispiel zeigt, wie mit einigen wenigen Funktionen ein Bridgespiel aufgebaut werden kann. Das Programm mischt die Karten und teilt sie an vier Spieler aus. Die Spieler können nun über die Punktwerte der Karten entscheiden, was ein Blatt wert ist. Dabei erhält das Ass vier Punkte, der König drei, die Königin zwei und der Bube einen. Das Programm simuliert das Mischen, Austeilen und Bewerten des Blattes.

Die Kartenstruktur und das Kartenarray des Spiels werden extern definiert und damit auch gleich initialisiert.

```
struct karten
{
   int augen;
   char farbe;
}
/* jede Karte belegt den kleinstmöglichen
Platz */
/* Spiel deklarieren und initialisieren */
struct karten blatt[4][13], deck[52]=
   {{1,'C'}, {2,'C'}, {3,'C'}, {4,'C'}, {5,'C'}, {6,'C'},
   {7,'C'}, {8,'C'}, {9,'C'}, {10,'C'}, {11'C'},
   {12'C'}, {13'C'}, {1,'D'}, {2,'D'}, {3,'D'}, {4,'D'},
   {5,'D'}, {6,'D'}, {7,'D'}, {8,'D'}, {9,'D'}, {10,'D'},
   {11,'D'}, {12,'D'}, {13,'D'}, {1,'H'}, {2,'H'},
   {3,'H'}, {4,'H'}, {5,'H'}, {6,'H'}, {7,'H'}, {8,'H'},
   {9,'H'}, {10,'H'}, {11,'H'}, {12,'H'}, {'H'13'H'},
   {1,'S', {2,'S', {3,'S', {4,'S', {5,'S', {6,'S', {7,'S',
   {8,'S', {9,'S', {10,'S', {11,'S', {12,'S', {13,'S'}}
   shuffle(spiel)
   struct karten spiel[];
   /* Wir setzen voraus, dass ein Zufallszahlengenerator – random(n) – vorhanden ist, der eine Zahl zwischen 0 und n-1 liefert. Die Systemfunktion rand() erzeugt eine Ganzzahl. */
   /* Die Karten werden durch zufallsgesteuertes Austauschen ihrer Position gemischt */
   {
      int i,j;
      for (i=0; i<52;++i)
      {
         j=random(52);
         swap(&spiel[i], &spiel[j];
      }
}swap (p,q)
struct karten*p, *q;
{
   struct karten temp;
   temp=*p;
   *p=*q
   *q=temp;
}
austeilen (spiel,blatt)
struct karten spiel[], blatt[][];
{  int i,j,k=0;
   for (i=0;i<4;++i)
      for (j=0;j<13;++j)
         blatt[i][j]=spiel[k++];
}
punkte_zaehlen(a_blatt)
struct karten *a_blatt
/* a_blatt ist ein Zeiger auf die erste Karte eines Blattes (ein Set von 13), z. B. &blatt[2][0] */
{
   int i, punkte_zaehlen=0;
   for (i=0;i<13;++i)
   {     if (a_blatt-)augen=1)
         punkte_zaehlen+=4;
      else if (a_blatt-)augen>10)
         punkte_zaeh-
            len+=a_blatt->augen
         -10;
      ++a-blatt;
/* zeigt zur naechsten Spielkarte */
   }
 return(punkte_zaehlen);
}
```

Das Dateiprinzip

Bei vielen Sprachen gibt es Schwierigkeiten, das beste Verhältnis zwischen völlig freien und exakt definierten Befehlsformaten zu finden. In C wurde dieses Problem auf sehr praktische Weise gelöst.

Wie bei vielen modernen Programmiersprachen ist auch in C die Ein- und Ausgabe (E/A) nicht fest in die Sprache eingebaut, da umfassende E/A-Definitionen (wie etwa bei COBOL und FORTRAN) verhindern würden, daß Anwendungen sich auf andere Maschinen und Systeme übertragen lassen.

So kann COBOL ohne Spracherweiterungen (die außerhalb des Standards liegen) nicht einmal Zeichen direkt von der Tastatur lesen. Wenn jedoch überhaupt keine E/A definiert ist, lassen sich Programme auf unterschiedlichen Betriebssystemen und Maschinen nur unter Schwierigkeiten zum Laufen bringen. Die Tastatureingabe bei PASCAL, die in den einzelnen Sprachversionen verschieden ausgelegt wurde, ist dafür ein gutes Beispiel.

Das Modul stdio.h mit seinen vorgefertigten E/A-Definitionen auf hoher und maschinennaher Sprachebene sorgt dafür, daß C eine Art Standard erhält. Da der Inhalt dieser Routinen von einer Anwendung zur anderen variieren kann, verwendet man am besten nur die Funktionen auf hoher Ebene.

Die Standardfunktionen für Bildschirmausgabe (printf) und Tastatureingabe (scanf) wurden bereits erläutert. Es gibt eine ganze Reihe weiterer Funktionen zur Steuerung von Bildschirm, Tastatur, Speicher und anderen Geräten. C behandelt Dateien und Geräte gleich und vereinfacht damit das gesamte Konzept der E/A. Für C ist eine Datei ein Bytestrom, aus dem sich die Eingabefunktionen ein Byte oder eine Bytegruppe herausnehmen und auf dem die Ausgabefunktionen ihre Daten aussenden. Ist der Strom mit einer Diskette verbunden, dann kann man damit Adressen ansprechen und wahlfrei darauf zugreifen. Der logische Unterschied zwischen Datei und Gerät besteht somit nicht mehr.

Normalerweise müssen Dateien vor ihrem Einsatz eröffnet werden. In jedem C-Programm sind jedoch drei Ströme ständig offen: stdin (normalerweise an die Tastatur angeschlossen), stdout und stderr (normalerweise mit dem Bildschirm verbunden). In vielen Betriebssystemen (z. B. Unix) läßt sich die Ein- und Ausgabe umkehren und so die Zuordnung verändern. Die Funktionen printf und scanf arbeiten mit stdout und stdin, sind aber nur Spezialfälle von fprintf und fscanf.

```
printf(. . . .);
fprint(stdout, . . . .);
```

haben die gleiche Wirkung. Zwei ähnliche Funktionen – sprintf und sscanf – führen im Hauptspeicher E/A-Vorgänge mit Strings aus.

Der Pointer stdout zeigt auf ein Datenelement vom Typ FILE, das normalerweise über ein Makro (#define) in stdio.h definiert ist. Die Eröffnung einer Datei (für eine Diskette oder ein Gerät) erledigt die Funktion fopen (dateiname, dateiart). Der String dateiname kann sich je nach Betriebssystem auf ein Gerät oder eine Datei beziehen. dateiart kann „r" für Lesen (aus der Datei), „w" für Schreiben oder „a" für das Anhängen von Daten (an die Datei) sein. Wenn eine mit „a" oder „w" eröffnete Datei nicht existiert, wird sie neu angelegt. Bereits bestehende, mit „w" eröffnete Dateien werden überschrieben. Der Dateipointer (eine Ganzzahl vom Typ „long integer", die die aktu-

Ernstfall

Es gibt viele Programmbibliotheken mit interessanten Funktionen und Makros, auf die wir aus Platzgründen nicht eingehen können. In der Datei ctype.h finden Sie eine Reihe Funktionen zur Bearbeitung einzelner Zeichen. Die Datei läßt sich mit
#include (ctype.h)
in ein Programm einschließen. Die erste Tabelle zeigt, wie getestet werden kann, ob Zeichen bestimmte Eigenschaften haben. Ein Wert ungleich Null bedeutet, daß das Zeichen die Eigenschaft besitzt (true).

Name	‚True' falls
isalpha(c)	c = Buchstabe
isupper(c)	c = Großbuchstabe
islower(c)	c = Kleinbuchstabe
isdigit(c)	c = Zahl
isxdigit(c)	c = Hexadezimalzahl
isspace(c)	c = Leerzeichen
isalnum(c)	c = Buchstabe oder Zahl
ispunct(c)	c = Interpunktionszeichen
isprint(c)	c = druckbares Zeichen
iscntrl(c)	c = Steuerzeichen
isascii(c)	c = ASCII-Code

Die zweite Funktionsgruppe führt Umwandlungen aus:

Name	wandelt
toupper(c)	c in Großbuchstaben
tolower(c)	c in Kleinbuchstaben
toascii(c)	c in ASCII-Code

Characterbewertung

```
/* Dieses Programm zaehlt die Zahl der Worte
und Zeichen in der Datei, deren Name in der
Befehlszeile angegeben ist */

# include(stdio.h)
# include(ctype.h)
main(argc,argv)
int argc;
char *argv;
  {
  int char_count=0, word_count=0,c,
  inword =0;
  FILE *in_file, *fopen();
/* Beachten Sie, dass der Dateiname und die
Funktion fopen als Pointer vom Typ FILE dekla-
riert sind */
/* Testen, ob die korrekte Argumentenzahl vor-
handen ist */
  If (argc !=2)
  {
    fprintf(stderr,"/nusage is %s filename/n",
    *argv);
/* Denken Sie daran, dass der Dateiname der
erste Eintrag im Array argv ist */
    exit (1);
  }
/* Datei eroeffnen und testen, ob sie existiert.
++argv zeigt auf den Dateinamen */
  if ((in_file=fopen(*++argv,"r"))==NULL)
{
    fprint(stderr,"/ncannot open &s/n",
    *argv);
/* Denken Sie daran, dass argv nun auf den
Dateinamen zeigt */
exit(1);
}
while ((c=getc(in_file)) !=EOF)
{
  ++char_count;
  if(inword)
  {
    if(isalnum(c))
    ;/* ein Leerbefehl */
    else
    {
      inword=0;
      ++word_count;
    }
  }
  else
    if (isalnum(c))
      inword=1;
}
if (inword)
  ++word_count;
printf("/nzeichenzahl=%d",char_count);
printf(/nzahl der worte=%d/n",word_count);
fclose(in_file);
}
```

elle Position im Bytestream angibt) wird an den Anfang einer mit „r" oder „w" eröffneten Datei gestellt. Die Funktion fopen liefert einen Pointerwert, den andere Funktionen verarbeiten können. Steht er auf Null, ist der Zugriff auf die Datei nicht möglich. Die Funktion fclose(pointer_auf_die_datei) schließt eine offene Datei.

E/A-Standardfunktionen

In der Standard-Programmbibliothek von C gibt es folgende weitere E/A-Funktionen:

getc(pointer_to_file) Holt das nächste Byte aus der Datei, die mit „r" eröffnet wurde. Das Ergebnis hat die Form einer Ganzzahl. Bei Dateiende erscheint ein spezieller EOF-Wert. Die Funktion läßt sich als Makro einsetzen.

getchar() Entspricht getc(stdin)

fgetc(pointer_to_file) Setzt das Zeichen c wieder in die Datei zurück, aus der es gelesen wurde. Das Ergebnis hat die Form einer Ganzzahl. Bevor das erste Zeichen in die Datei zurückgeschrieben werden kann, muß zumindest ein Zeichen von dort gelesen worden sein.

putc(c,pointer_to_file) Übergibt das Zeichen c an die Ausgabedatei und liefert den Ganzzahlenwert von c. Als Makro einsetzbar.

putchar(c) Entspricht putc(c,stdout)

fputc(c,pointer_to_file) Entspricht putc, wird aber nur als Funktion eingesetzt.

gets(s) Der String s ist Pointer auf char und liest bis zum nächsten Zeilenvorschub Zeichen aus stdin. Der Zeilenvorschub wird nicht mehr an den String übergeben und der String nun mit „/0" abgeschlossen. Zurückgegeben wird der Wert von s.

fgets(s,n,pointer_to_file) Liest Zeichen aus der Datei und speichert sie im String s, bis entweder n-1 Zeichen gelesen wurden oder der nächste Zeilenvorschub erscheint. Der Zeilenvorschub wird an s übergeben und s mit „/0" abgeschlossen. Zurückgegeben wird der Wert von s.

puts(s) Gibt den String s an stdout aus und hängt einen Zeilenvorschub an.

fputs(s,pointer_to_file) Gibt den String s ohne Zeilenvorschub an die Datei aus.

fseek(pointer_to_file,offset,place) Verschiebt den Dateipointer auf dem Bytestrom bis zum Offset oder von der angegebenen Position an um eine bestimmte Anzahl Bytes. Place kann folgende Werte haben: 0 für den Dateianfang, 1 für die aktuelle Position und 2 für das Dateiende. Offset sollte eine Ganzzahl vom Typ „long int" sein.

rewind(pointer_to_file) entspricht:

fseek(pointer_to_file,0L,0)

ftell(pointer_to_file) Liefert das aktuelle Offset (Typ „long int") vom Dateianfang aus.

unlink(filename) Löscht die angegebene Datei im Directory. Liefert -1, wenn es die Datei nicht gibt, und 0, wenn sie existiert.

exit(status) Beendet ein Programm und liefert den Ganzzahlenwert von Status an das Betriebssystem oder das aufrufende Programm. 0 zeigt ein normales Ende an.

In der Bibliothek

Wir beenden unsere C-Serie mit einem kurzen Blick auf die Schnittstelle zwischen C und dem Betriebssystem Unix.

Die Sprache C wird hauptsächlich zur Entwicklung von Systemsoftware, Betriebssystemen, Dienstprogrammen und schnellen Anwendungsprogrammen eingesetzt. Für diesen Zweck muß zwischen der Sprache und dem zugehörigen Betriebssystem eine Schnittstelle existieren, über die auch Systemfunktionen programmierbar sind.

Speziell die Funktion sizeof(objekt) unterstützt die Entwicklung von Code, der auf unterschiedlichen Systemen laufen soll. sizeof(objekt) wird mit einem Variablennamen oder einem Typennamen geladen und liefert die Anzahl Bytes, die zur Speicherung der Variablen nötig sind. Mit Hilfe von sizeof(objekt) kann der Programmcode unabhängig davon funktionieren, ob int 16 oder 32 Bits lang ist. sizeof(objekt) wird häufig mit den Speicherfunktionen calloc(), malloc(), realloc() und free() eingesetzt.

- **calloc():**

```
char *calloc(anzahl,laenge)
int anzahl,laenge
```

calloc() reserviert für anzahl Variablen einen Speicherbereich (laenge gibt die Bytezahl der Variablen an), initialisiert ihn auf Null und liefert einen Pointer auf das erste Datenelement (oder Null, wenn zu wenig Speicher vorhanden ist). Der folgende Befehl reserviert Platz für 50 Ganzzahlen:

```
p=calloc(50,sizeof(int));
```

- **malloc**

```
char *malloc(bytes)
unsigned bytes;
```

malloc() ähnelt calloc(), initialisiert die Bytes jedoch nicht. Beide Funktionen liefern einen Pointerwert auf char, da char nur ein Byte lang ist. Wenn er auf andere Datentypen zeigen soll, muß er umgewandelt werden.

- **realloc**

```
char *realloc(p,bytes)
char *p;
int bytes;
```

realloc() ändert die Größe des Speicherbereichs (auf den p zeigt) in die mit bytes angegebene Bytezahl und kopiert den Inhalt des alten Bereichs in den neuen Bereich. Da der Speicherbereich sich dabei verschieben kann, sind Pointer auf den alten Bereich nicht mehr gültig.

- **free():**

```
free(p)
char *p;
```

Diese Funktion gibt einen Speicherbereich frei, der mit calloc, malloc oder realloc reserviert wurde. Der Einsatz anderer Pointerwerte kann unerwünschte Folgen haben.

Eine weitere Kategorie von C-Funktionen steuert die Fehlerbehandlung. In C-Programmen treten die meisten Fehler bei der Ein- und Ausgabe oder beim Aufruf von Bibliothekfunktionen auf. Die Bibliothek enthält die Standardvariable errno vom Typ int und eine Liste von Fehlermeldungen namens sys_err-list[], die vom Programm aus direkt angesprochen werden kann. Wenn beim Ablauf von Bibliotheksfunktionen Fehler eintreten, wird errno mit einer Zahl geladen. Der Befehl

```
perror(s)
char *s="Fehlermeldung";
```

gibt auf den Datenstrom stderr je nach Inhalt von errno die Fehlermeldung des Systems und der Bibliothek aus.

Da bei Ein- und Ausgabefehlern nicht immer eine normale Fehlermeldung des Systems ausgegeben wird, testet int ferror(file_pointer), ob der Fehler beim Schreiben oder Lesen der angegebenen Datei aufgetreten ist. Die Funktion liefert Null (Fehler) oder ungleich Null (kein Fehler). Da Fehler den Dateizugang behindern können, läßt sich die Fehleranzeige auch mit der Funktion clearerr (file_pointer) löschen.

Eine Reihe weiterer Funktionen verwalten Diskettendateien und Directories. Das Handbuch des C-Compilers liefert hier alle nötigen Einzelheiten. Darunter gibt es mit Sicherheit den Befehl access(), der den Zugriffsmodus der angegebenen Datei oder des Directories prüft; chmod(), der den Zugriffsmodus ändert und chdir(), der ein anderes Directory aufruft.

Unser Programmbeispiel enthält alle Fähigkeiten von C, die wir bisher besprochen haben. Einige davon wurden zwar nur kurz erwähnt (z. B. rekursive Funktionsaufrufe), andere jedoch (z. B. malloc) ausführlich behandelt. Das Programm erstellt den Index einer langen Textdatei – das heißt eine Liste aller Wörter mit den entsprechenden Seitennummern. Die Wörter sind in einer verketteten Liste gespeichert, die mit jedem neuen Wort wächst. Jedem Wort ist ein Pointer zugeordnet, der auf eine weitere verkettete Liste mit den Seitennummern zeigt.

C ist eine der wenigen Sprachen, in der auf kleinen Micros entwickelte Programme auch auf großen Micros, Minis und sogar den Geräten der Groß-EDV laufen.

Indexaufbau

```
/* In der Datei index.h */
#define NULL 0
#define MAXWORDSIZE 20
typedef char ENTRY [MAXWORDSIZE];
/* Jedes Element des Indizes enthält den direkten
Eintrag, einen Pointer auf die Liste der Seiten-
nummern und einen Pointer auf das nächste
Datenelement */
struct page__number
{
    int pn;
    struct page__number *pnext;
}
typedef struct page__number page;
typedef page *plink;
struct index__element
{
    ENTRY entry;
    plink pages;
    struct index__element *exit;
}
typedef struct index__element element;
typedef element *link;
/* Wir können uns nun mit ‚element' auf ein
Listenelement beziehen. Ein Pointer auf ein Ele-
ment wird ‚link' genannt. */
/* In der Datei index.c */
#include stdio.h
#include string.h
#include index.h
#define LPP = 66; /* lines per page */
main(argc,argv)
int argc;
char *argv[];
link head;
{
    FILE infile;
    int lc = 0, pc = 1, inword = 0;
    ENTRY nextword;
    char *nw;
/* Liste der Einfachheit halber mit einem ersten
Dummyeintrag füllen */
    head = new__entry ("",NULL, 0);
/* Dateinamen in den Index eintragen */
    if(argc != 2)
    {
        fprintf (stderr, "\nusage is %s filename\n",
        *argv);
        exit(1);
    }
    if (infile = fopen (*++argv, "r") == NULL)
        fprintf(stderr, "\nfile not found %s\n",
        *argav);
        exit(1);
    }
    nw = nextword;
    while (c = getc(infile) != EOF)
    {
        if(c == '\n')
/* Eins zu Zähler addieren, prüfen, ob Seitenende */
        {
            lc += 1;
            if (lc> LPP)
            {
                pc += 1;
                lc = 0;
            }
        }
        else if (inword)
        {
            if (isalpha(c))
                *nw++ = c;
            else
            {
                inword = 0;
                *nw = '\o';
                insert (nextword, head, pc);
                nw = nextword;
            }
        }
        else
        {
            if (isalpha(c))
            {
                inword = 1;
                *nw++ = c;
            }
        }
    }
    if(inword)
    {
        *nw = '\o';
        insert (nextword, head, pc);
    }
    display__index;
}
insert(e)
ENTRY e;
{
insert(e,l,pnum)
ENTRY e;
link l;
int pnum;
static link lastl;
{
    lastl = head;
    if(l == NULL)
    {
        lastl—>next = new__entry (e,l,pnum);
        return;
    }
    else
    {
        int s;
        s = strcmp(e,l—>entry);
        if(s == 0)
/* Wort ist vorhanden, Seitennummer hinzufügen */
        {
            add__page__number (pnum,
            l—>pages);
            return;
        }
        else if(s > 0)
/* zu weit, daher nach dem letzten Datenelement
ein neues einsetzen */
        {
            lastl—>next = new__entry (e,l,pnum);
            return;
        }
        else
        {
/* noch nicht gefunden, daher mit einem rekur-
siven Modul Liste für das Einsetzen durchsuchen */
            lastl = l;
            insert(e,l—>next, pnum);
            return;
        }
    }
}
link new__entry(e,l,pnum)
ENTRY e;
link l;
int pnum;
{
/* mit malloc Platz für Eintrag schaffen */
    link newl;
    newl = (link) malloc (sizeof (element));
/* Umwandlung des von malloc gelieferten char
Pointers */
    newl—>entry = e;
    newl—>next = l;
    newl—>pages = (plink) malloc (sizeof (page));
    newl—>pages—>pn = pnum;
    newl—>pages—>pnext = NULL;
    return(newl);
}
add__page__number (pnum,pl)
int pnum;
plink pl;
{
/* Ende der Liste der Seitennummern finden */
    while (pl—>pnext) != NULL)
        pl = pl—>pnext;
    pl—>pnext = (plink) malloc (sizeof (page));
    pl—>pnext—>pnext = NULL;
    pl—>pnext—>pn = pnum;
    return;
}
display__index ;
link l;
plink pl;
{
    l = head—>next;
    while(l != NULL)
    {
        printf ("%s\t", l—>entry) ;
        pl = l—>pages;
        while (pl—>next != NULL)
        {
            printf("%d4,",pl—> pn);
            pl = pl—>next;
        }
        printf("%d\n", pl—>pn);
    }
    return;
}
```